

GuLoader Obfuscation Analysis | ThreatLabz

zscaler.com/blogs/security-research/technical-analysis-guloader-obfuscation-techniques

ThreatLabz

February 9, 2026



Technical Analysis

This section covers the obfuscation techniques that GuLoader leverages to hinder analysis and evade detection.

Dynamic constant construction

GuLoader employs polymorphic code to dynamically construct constants during execution. Instead of embedding these values statically, GuLoader uses a combination of assembly operations such as `mov`, `xor`, `add`, and `sub` to build the constants as needed, as shown in the figure below.

```

mov     [ebp+1D1h], ecx
mov     ecx, 0D186CF5Ch
xor     ecx, 14680630h
cmp     eax, 728AA914h
xor     ecx, 9F0CEA6Fh
sub     ecx, 5AE222A2h ; ((0xD186CF5C ^ 0x14680630) ^ 0x9F0CEA6F) - 0x5AE222A2 = 0x61
test    bh, ch
cmp     ebx, ecx
mov     ecx, [ebp+1D1h]
jnl     short loc_645AFE
cmp     cx, 17FDh
mov     dword ptr [ebp+236h], 457BC0B3h
sub     dword ptr [ebp+236h], 0FFC61E7Fh
xor     dword ptr [ebp+236h], 87B5AECEh
cmp     ax, bx
add     dword ptr [ebp+236h], 3DFFF380h ; ((0x457BC0B3 - 0xFFC61E7F) ^ 0x87B5AECE) + 0x3DFFF380 = 0x7a
test    esi, 701C9553h
cmp     ebx, [ebp+236h]
jg      short loc_645AFE
test    bh, ah
mov     [ebp+1F0h], eax
mov     eax, 0B3BE5AE4h
xor     eax, 32CF27C1h
test    ch, bh
cmp     bh, 0ABh
xor     eax, 0FC7615DFh
clc
sub     eax, 7D0768DAh ; ((0xB3BE5AE4 ^ 0x32CF27C1) ^ 0xFC7615DF) - 0x7D0768DA = 0x20
cmp     dl, bl
sub     ebx, eax

```



Figure 1: Shows an example of the operations that GuLoader uses to dynamically construct constant values during execution.

The main purpose of obfuscating these constant values is to increase the difficulty of interpreting the underlying code. The polymorphic operations also impede static-based signatures that can be used for detection.

Exception-based code redirection

GuLoader utilizes a control flow obfuscation technique that replaces standard code jump (`jmp`) instructions with deliberate CPU exceptions.

- **Exception handling:** GuLoader sets up a custom exception handler designed to intercept and process designated exceptions.
- **Intentional exception:** Rather than using a standard jump instruction, GuLoader executes carefully crafted instructions intended to deliberately trigger specific exceptions.
- **Code redirection:** Upon activation, the exception handler calculates the correct destination address and modifies the instruction pointer to continue execution at the intended location.

This technique makes the malware's execution flow extremely difficult for automated analysis tools to trace. The table below outlines the exception types that GuLoader actively handles across different versions.

Exception Code	Exception Type	2022	2023	2024-2025
0x80000003	STATUS_BREAKPOINT	X	X	X
0x80000004	STATUS_SINGLE_STEP		X	X
0xC0000005	STATUS_ACCESS_VIOLATION		X	X
0xC000001D	STATUS_ILLEGAL_INSTRUCTION			X
0xC0000096	STATUS_PRIVILEGED_INSTRUCTION			X

Table 1: Exception types handled by GuLoader across different versions.

Software breakpoint exceptions

Early versions of GuLoader implemented a simplified approach to exception-based control flow obfuscation. The malware would trigger a software breakpoint interrupt by executing an `int 3` instruction, as shown in the figure below.

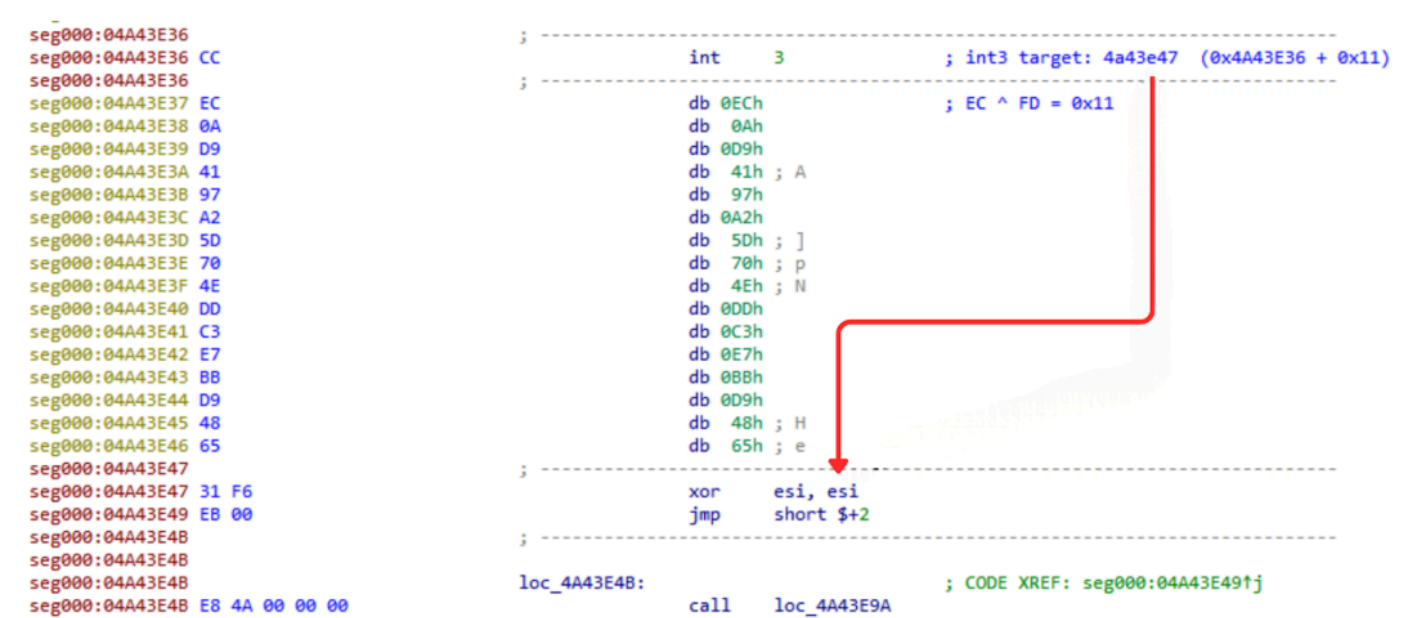


Figure 2: Demonstrates version 2022 of GuLoader's use of an `int 3` instruction to trigger a software interrupt.

GuLoader's custom exception handler would then take control after the `int 3` command was triggered. The handler analyzed the byte of data located immediately after the interrupt instruction, performed a simple calculation (an XOR operation, where the XOR key remains the same across all operations), and derived the actual destination address for the jump. Once calculated, the handler redirected the program's execution, resuming it at the intended location, as shown in the figure below.

```

seg000:04A52DE3 88 44 24 04
seg000:04A52DE7 84 C2
seg000:04A52DE9 8B 10
seg000:04A52DEB 8B 12
seg000:04A52DED 81 FA 03 00 00 80
seg000:04A52DF3 0F 85 4C 01 00 00
seg000:04A52DF9 8B 40 04
seg000:04A52DFC 83 78 04 00
seg000:04A52E00 0F 85 3F 01 00 00
seg000:04A52E06 83 78 08 00
seg000:04A52E0A 0F 85 35 01 00 00
seg000:04A52E10 83 78 0C 00
seg000:04A52E14 0F 85 28 01 00 00
seg000:04A52E1A 83 78 10 00
seg000:04A52E1E 0F 85 21 01 00 00
seg000:04A52E24 83 78 14 00
seg000:04A52E28 0F 85 17 01 00 00
seg000:04A52E2E 83 78 18 00
seg000:04A52E32 0F 85 0D 01 00 00
seg000:04A52E38 66 81 F9 6D FD
seg000:04A52E3D 8B 90 B8 00 00 00
seg000:04A52E43 EB 34

@EXCEPTION_HANDLER_INT3:
; !!
mov     eax, [esp+exception_pointers]
test    dl, al
mov     edx, [eax] ; edx -> EXCEPTION_RECORD
mov     edx, [edx] ; edx = exception code
cmp     edx, 80000003h
jnz     int3_handler_exit
mov     eax, [eax+4] ; eax -> CONTEXT
cmp     [eax+CONTEXT.Dr0], 0
jnz     int3_handler_exit
cmp     [eax+CONTEXT.Dr1], 0
jnz     int3_handler_exit
cmp     [eax+CONTEXT.Dr2], 0
jnz     int3_handler_exit
cmp     [eax+CONTEXT.Dr3], 0
jnz     int3_handler_exit
cmp     [eax+CONTEXT.Dr6], 0
jnz     int3_handler_exit
cmp     [eax+CONTEXT.Dr7], 0
jnz     int3_handler_exit
cmp     cx, 0FD6Dh
mov     edx, [eax+CONTEXT.Eip]
jmp     short loc_4A52E79

loc_4A52E79:
test    dx, ax
cmp     byte ptr [edx], 0CCCh
jnz     int3_handler_exit
mov     dl, [edx+1] ; byte(int3_addr + 1)
cmp     ax, 0B8E2h
xor     dl, 0FDh ; byte(int3_addr + 1) ^ 0xfd
test    dx, 973Eh
movzx   edx, dl
mov     ecx, [eax+CONTEXT.Eip]
test    edx, edx
add     ecx, edx ; ecx = int3_addr + (byte(int3_addr + 1) ^ 0xfd)
cmp     bl, al
dec     ecx

```

Figure 3: Example of GuLoader's exception handler observed in samples from 2022.

In early initial versions of GuLoader, the exception handler included an additional anti-debugging mechanism that verified the presence of software breakpoints at the address of the jump destination. This extra step added another layer of complexity and made analysis even more complex. However, this feature was removed in later versions of GuLoader. The figure below depicts GuLoader 2022's mechanism for scanning software breakpoints that check for the value 0xCC (i.e., `int 3`).

```

seg000:04A52EA4      53
seg000:04A52EA5  8B 98 88 00 00 00
seg000:04A52EAB  83 C3 02
seg000:04A52EAE  39 C8
seg000:04A52EB0  74 19
seg000:04A52EB2  66 39 C1
seg000:04A52EB5  5B
seg000:04A52EB6  38 FD
seg000:04A52EB8  80 39 CC
seg000:04A52EBB  0F 84 84 00 00 00
seg000:04A52EC1  38 F7
seg000:04A52EC3  49
seg000:04A52EC4  84 EC
seg000:04A52EC6  EB DC

scan_breakpoints_in_code_to_jump:
    push    ebx
    mov     ebx, [eax+CONTEXT._Eip]
    add     ebx, 2
    cmp     ebx, ecx
    jz      short breakpoints_not_found_continue_ok
    cmp     cx, ax
    pop     ebx
    cmp     ch, bh
    cmp     byte ptr [ecx], 0CCh
    jz      int3_handler_exit
    cmp     bh, dh
    dec     ecx
    test    ah, ch
    jmp     short scan_breakpoints_in_code_to_jump

breakpoints_not_found_continue_ok:
    cmp     cx, bx
    pop     ebx
    cmp     ax, bx
    add     [eax+CONTEXT._Eip], edx ; continue execution at
    jmp     short loc_4A52F16 ; int3_addr + (byte(int3_addr + 1) ^ 0xfd)

seg000:04A52ECB  66 39 D9
seg000:04A52ECE  5B
seg000:04A52ECF  66 39 D8
seg000:04A52ED2  01 90 B8 00 00 00
seg000:04A52ED8  EB 3C

```

Figure 4: Example of GuLoader manipulating control flow via software breakpoints.

In 2023, GuLoader's exception handler was updated to support two additional exceptions: `0x80000004` (`STATUS_SINGLE_STEP`) and `0xC0000005` (`STATUS_ACCESS_VIOLATION`). For these two exceptions, the exception handler follows an approach similar to software breakpoint exceptions. However, in this case, the (encrypted) jump offset is located two bytes past the exception address.

```

v1 = **retaddr;
switch ( v1 )
{
    case 0xC0000005:
        if ( (*retaddr)[6] > 0x10000 )
            return 0;
LABEL_13:
    v7 = ANTI_CheckDrRegs();
    *(_DWORD *) (v7 + 0x88) += *(_BYTE *) (*(_DWORD *) (v7 + 0x88) + 2) ^ 0x29; // Encrypted offset = exception EIP + 2
    return 0xFFFFFFFF;
    case 0x80000004:
        goto LABEL_13;
    case 0x80000003:
        v2 = ANTI_CheckDrRegs();
        v3 = *(_BYTE **) (v2 + 0x88);
        if ( *v3 == 0xCC )
        {
            // Encrypted offset = exception EIP + 1
            v4 = v3[1] ^ 0x29;
            for ( i = (_BYTE *) (v4 + *(_DWORD *) (v2 + 0x88) - 1); (_BYTE *) (*(_DWORD *) (v2 + 0x88) + 2) != i; --i )
            {
                if ( *i == 0xCC )
                {
                    // Scan for breakpoints in the next instructions
                    return 0;
                }
            }
            *(_DWORD *) (v2 + 0x88) += v4;
            return 0xFFFFFFFF;
        }
        break;
}
return 0;

```



Figure 5: Example of GuLoader's exception handler observed in samples from 2023.

Single step exceptions

GuLoader purposefully triggers a single step exception (0x80000004) by manipulating the EFLAGS register by using the PUSHF instruction to copy the current EFLAGS onto the stack. GuLoader enables the Trap Flag (TF) by setting bit 8 of the EFLAGS value by adding the EFLAGS value on the stack with 0x100. The result (with the TF flag set) is then written to the EFLAGS register by executing a POPF instruction. When the very next instruction is executed by the CPU, the single step exception will be triggered and processed by GuLoader's exception handler. The example below shows how GuLoader triggers a single step exception and how the exception handler (shown in the previous figure) redirects the control flow to the next valid instruction.

```

.data:004032A1      mov     ecx, 0A491237Eh
.data:004032A6      xor     ecx, 26A5C9A4h
.data:004032AC      xor     ecx, 246D6D47h
.data:004032B2      sub     ecx, 8EBB311h
.data:004032B8      sub     ecx, 9D6DD38Ch ; ecx = 0x100  TF Trap Flag (single step)
.data:004032BE      push    esi
.data:004032BF      pushf                    ; trick (type_PUSHF) target: 4032d1 (off: a)
.data:004032C0      mov     esi, esp
.data:004032C2      add     [esi], ecx
.data:004032C4      popf
.data:004032C5      cmp     eax, edi
.data:004032C5 ; -----
.data:004032C7      db     75h ; u
.data:004032C8      db     5
.data:004032C9      db     23h ; # ; 0x23 ^ 0x29 = 0x0a (offset jump)
.data:004032CA      db     13h
.data:004032CB      db     0B5h
.data:004032CC      db     9Dh
.data:004032CD      db     0FBh
.data:004032CE      db     25h ; %
.data:004032CF      db     66h ; f
.data:004032D0      db     0
.data:004032D1 ; -----
.data:004032D1      test    dx, cx
.data:004032D4      pop     esi
.data:004032D5      cmp     bh, dh
.data:004032D7      pop     ecx
.data:004032D8      push   edi
.data:004032D9      mov     edi, 84FD6C11h
.data:004032DE      add     edi, 3C344A77h

```

Figure 6: Example of GuLoader code leveraging single-step exceptions to manipulate control flow.

Access violation exceptions

GuLoader intentionally attempts to access (e.g., write to) a memory address below 0x10000, triggering an access violation. The custom exception handler intercepts this error and redirects the instruction pointer to the intended destination, as shown in the figure below.

```

.data:00403256      push    ebx
.data:00403257      mov     ebx, 752F6110h
.data:0040325C      xor     ebx, 0B33C84C4h
.data:00403262      sub     ebx, 147E91C8h
.data:00403268      xor     ebx, 8D49A353h
.data:0040326E      add     ebx, 0C32349C7h ; ((0x752f6110 ^ 0xb33cb4c4) - 0x147e91c8) + 0xc32349c7 = 0x3126
.data:00403274      mov     [ebx], edx ; trick (type_MOV) target: 40327e (off: a)
.data:00403274 ; -----
.data:00403276      db     23h ; # ; 0x23 ^ 0x29 = 0x0a (offset jump)
.data:00403277      db     89h
.data:00403278      db     0FFh
.data:00403279      db     0DDh
.data:0040327A      db     9
.data:0040327B      db     43h ; C
.data:0040327C      db     0D9h
.data:0040327D      db     0
.data:0040327E ; -----
.data:0040327E      pop     ebx
.data:0040327F      push    4F3B5615h

```

Figure 7: Example of GuLoader code leveraging access violation exceptions to manipulate control flow.

In 2024, GuLoader introduced support for two new additional exceptions: *0xC000001D* (*STATUS_ILLEGAL_INSTRUCTION*) and *0xC0000096* (*STATUS_PRIVILEGED_INSTRUCTION*). These changes both led to a more intricate method for calculating the jump address. Since the instructions that trigger these exceptions can vary in length, placing the jump offset directly after the instruction is unreliable. To solve this, GuLoader's developers implemented a fixed, hardcoded offset within the exception handler. This offset consistently locates the encrypted jump address, regardless of the preceding instruction's size. Rather than relying on a single obfuscated byte, the updated handler now includes a hardcoded offset that points to a secondary byte. This secondary byte contains the encrypted offset to the address of the jump target. To decrypt the encrypted offset, the handler uses a dynamically generated XOR key that ultimately reveals the final jump destination. This multi-step approach significantly increases the complexity of the technique, making the jumps even harder to trace. For example, in the figure below, the hardcoded offset is 0x23, and the XOR key used to decrypt this offset is 0x85. This dynamically generated XOR key is created within the same subfunction of the exception handler that also verifies hardware breakpoints, as shown in the figure below.

```

unsigned __int32 __stdcall exception_handler(PEXCEPTION_POINTERS ExceptionInfo)
{
    NTSTATUS ExceptionCode; // edx
    infoctx exception_info; // kr00_8

    ExceptionCode = ExceptionInfo->ExceptionRecord->ExceptionCode;
    if ( ExceptionCode != STATUS_ACCESS_VIOLATION )
    {
        if ( ExceptionCode != STATUS_ILLEGAL_INSTRUCTION
            && ExceptionCode != STATUS_PRIVILEGED_INSTRUCTION
            && ExceptionCode != STATUS_SINGLE_STEP
            && ExceptionCode != STATUS_BREAKPOINT )
        {
            return 0;
        }
    }
    handle_exception:
    exception_info = (check_for_hw_bps_obtain_eip_xor_key)(ExceptionInfo);
    get_next_eip_address(&exception_info.context->Eip, (exception_info.context->Eip + 0x23), exception_info.eip_xor_key);
    return _byteswap_ulong(0xFFFFFFFF);
}
if ( ExceptionInfo->ExceptionRecord->ExceptionInformation[ACCESS_VIOLATION_ADDRESS] <= 0x10000 )
    goto handle_exception;
return 0;
}

```

Figure 8: Example of GuLoader 2024 leveraging the five different exception types progressively added across versions to manipulate control flow.

Dynamic hashing

Similar to many malware families, GuLoader uses the DJB2 hashing algorithm to identify API functions, modules, and process names. The GuLoader versions released after 2022 combine the DJB2 hash value with a bitwise XOR operation and a hardcoded 32-bit value (DWORD). The result is then compared against a pre-calculated list of expected hash values. This post-hash step is also common in malware families to prevent static values that can be used to create static detections.

Encrypted strings

GuLoader hides its command-and-control (C2) domains, file paths, and other critical information by encrypting strings with a simple XOR algorithm. Although the encryption mechanism itself is basic, the real challenge complexity lies in GuLoader's polymorphic code, which makes the strings difficult to locate and decrypt.

Static encrypted strings

In version 2022, GuLoader stored encrypted strings statically within shellcode, along with the corresponding string decryption key, as shown in the figure below.

```

seg000:04A5080C 66 39 C2
seg000:04A5080F 68 B7 49 BB 34
seg000:04A50814 38 CA
seg000:04A50816 81 2C 24 83 48 77 06
seg000:04A5081D 81 34 24 91 7B EC 34
seg000:04A50824 66 85 C0
seg000:04A50827 81 2C 24 69 7A A8 1A
seg000:04A5082E EB 40

cmp dx, ax
push 348B49B7h
cmp dl, cl
sub [esp+4+var_4], 6774883h
xor [esp+4+var_4], 34EC7B91h
test ax, ax
sub [esp+4+var_4], 1AA87A69h
jmp short loc_4A50870

loc_4A50870:
cmp ax, bx ; CODE XREF: sub_4A5080C+22↑j
push dword ptr [ebp+140h]
push dword ptr [ebp+14Ch]
call simple_xor_bufs ; xor given bytes with key from 0x4a50beb
cmo eax, ebx

...
call sub_4A5080C
; END OF FUNCTION CHUNK FOR decrypt_str
; -----
db 0C7h ; !! encrypted strings key
db 0E9h
db 8Eh
db 83h
db 61h ; a
db 24h ; $
db 3Dh ; =
db 0BEh
db 0A7h

```

← Leverages the return address from the CALL instruction to obtain the pointer to the encryption key.

Figure 9: Version 2022 of GuLoader's string decryption.

GuLoader used a clever technique to handle encrypted strings. A **CALL** instruction was placed immediately before the string's XOR key to push the key's memory address onto the stack. The XOR key size is then dynamically calculated and written to the stack. In the example above, the size value is calculated using the formula $((0x34BB49B7 - 0x6774883) \wedge 0x34EC7B91) - 0x1AA87A69 = 0x3C$. Similarly, another **CALL** instruction was used to push the memory address of the encrypted string onto the stack, as shown in the figure below.

```

seg000:04A4E149
seg000:04A4E149 68 01 56 34 19
seg000:04A4E14E CC
seg000:04A4E14F F4
seg000:04A4E14F
seg000:04A4E14F
seg000:04A4E150 AA
seg000:04A4E151 86
seg000:04A4E152 83
seg000:04A4E153 0E
seg000:04A4E154 9B
seg000:04A4E155 50
seg000:04A4E156 91
seg000:04A4E157
seg000:04A4E157 81 04 24 49 9C F9 C0
seg000:04A4E15E 81 34 24 BD FB E4 83
seg000:04A4E165 81 04 24 09 F6 36 A6
seg000:04A4E16C CC
seg000:04A4E16D EC
seg000:04A4E16E BC FF 32 02 87
seg000:04A4E173 90
seg000:04A4E174 63 A8 4D 54 BD B1
seg000:04A4E17A 04 3A
seg000:04A4E17A
seg000:04A4E17C B6
seg000:04A4E17D
seg000:04A4E17D E8 96 26 00 00
seg000:04A4E17D

sub_4A4E149 proc near
; CODE XREF: seg000:loc_4A4E8D1ip
push 19345601h
int 3
hlt
endp

sub_4A4E149
;
db 0AAh
db 86h
db 83h
db 0Eh
db 9Bh
db 50h ; P
db 91h
;
add dword ptr [esp], 0C0F99C49h
xor dword ptr [esp], 83E4F8BDh
add dword ptr [esp], 0A636F609h ; ((0x19345601 + 0xc0f99c49) ^ 0x83e4fbbd) + 0xa636f609 = 0
int 3 ; int3 target: 4a4e17d
in al, dx
mov esp, 870232FFh
nop
arpl [eax-4E42AB83h], bp
add al, 3Ah ; ':'
;
db 006h
;
call decrypt_str

loc_4A4E8D1:
call sub_4A4E149 ; CODE XREF: seg000:04A4E144tj
;
db 40h ; @
db 81h
db 89h
db 41h ; A
db 004h
db 81h
db 0EBh
db 0EFh
db 00h

```

Figure 10: Shows version 2022 of GuLoader's string decryption process.

Additionally, a value is dynamically calculated and pushed onto the stack after the address of the encrypted string, indicating whether the string is ASCII (value 0) or wide (value 1). In the example above, the value is 0 to denote that the string is ASCII. Finally, the malware invoked the decryption function `simple_xor_bufs` (found inside the function `decrypt_str`), which retrieved both the encrypted string address and decryption key address from the stack to perform the XOR operation to obtain the final string.

The first four bytes (DWORD) of each encrypted string encoded the string's length. The size is decrypted using a separate 4-byte XOR key.

[GitHub](#) repository, to decrypt the static encrypted strings found in GuLoader samples from 2022.

Stack-based string encryption

Starting in 2023, GuLoader updated their string encryption algorithms to use more convoluted polymorphic code that dynamically constructs the decrypted string with a combination of `mov`, `xor`, `add`, and `sub` operations on hardcoded constants, as shown in the figure below.

The screenshot displays the IDA Pro interface. On the left, the 'Functions' window lists various functions, including several 'CORE_build_enc_string' functions. The main window shows the pseudocode for a function named 'Pseudocode-A'. The code is as follows:

```

32  *retaddr = 0xCC3E5AC0;
33  *retaddr ^= 0x1263C601u;
34  *retaddr ^= 0x34C616CAu;
35  *retaddr += 0x15647646;
36  *(_DWORD *)((char *)a2 + 0x22F) = a1;
37  v4 = (char *) (retaddr + 1);
38  retaddr[1] = 0x3BEF3B29;
39  retaddr[1] ^= 0x2C25147Bu;
40  retaddr[1] += 0x248EFBE5;
41  retaddr[1] += 0x4DA2C3A;
42  a2[0x85] = 0x1B8B99CF;
43  a2[0x85] ^= 0x8803F308;
44  a2[0x85] += 0x3A05316C;
45  a2[0x85] ^= 0xCDAD9C30;
46  do
47  {
48      ++v4;
49      v5 = a2[0x85]-- == 1;
50  }
51  while ( !v5 );
52  v6 = v4 + 1;
53  *v6 = 0x9E11AA0A;
54  *v6 ^= 0x16CEC551u;
55  *v6 ^= 0xE4AB9480;
56  *v6++ ^= 0xC8C62BF3;
57  *v6 = 0xAE714921;
58  *v6 ^= 0x2F24CC53u;
59  *v6 += 0x27B019D2;
60  *v6 += 0x1BC52BA3;
61  *(_DWORD *)((char *)a2 + 0x1E1) = v24;
62  ++v6;
63  v7 = *(_DWORD *)((char *)a2 + 0x1E1);
64  *v6 = 0x73C1C5F6;
65  *v6 ^= 0x1E771E22u;
66  *v6 ^= 0xE3040607;
67  *v6 += 0x1E8B20E;
68  v8 = v6 + 1;
69  *v8 = 0xDDD9F122;
70  *v8 ^= 0x7F85B60Fu;
71  *v8 -= 0x96017AE;
72  *v8 ^= 0xA48E4D36;
73  *(_DWORD *)((char *)a2 + 0x22B) = 0x7C9EC0A6;
74  *(_DWORD *)((char *)a2 + 0x22B) ^= 0x80B3F028;
75  *(_DWORD *)((char *)a2 + 0x22B) ^= 0x44F440FBu;
76  *(_DWORD *)((char *)a2 + 0x22B) += 0x47268F8E;

```

Figure 11: Example of a GuLoader function utilizing polymorphic code to dynamically decrypt a string on the stack.

Once the individual components of the encrypted string and the encryption key are constructed, GuLoader uses simple XOR operations to decrypt the string. This modification to the string algorithm was further designed to complicate analysis and detection efforts, making emulation one of the best approaches to obtain the decrypted string.

Payload decryption

One of GuLoader's encrypted strings is binary data, often exceeding 0x300 bytes in length. This binary buffer functions as an XOR key, which is used to decrypt a malware payload that is downloaded from a hardcoded URL. The payload's URL, which is itself an encrypted string, often points to a shared file hosted on legitimate cloud services like Google Drive or OneDrive.

IDA scripts

To effectively deobfuscate GuLoader's constants, strings, and control flow, ThreatLabz created IDA scripts that are available in the ThreatLabz [GitHub](#) repository. These scripts dynamically calculate constants and string values, as well as remove the exception-based control flow obfuscation to streamline code analysis.